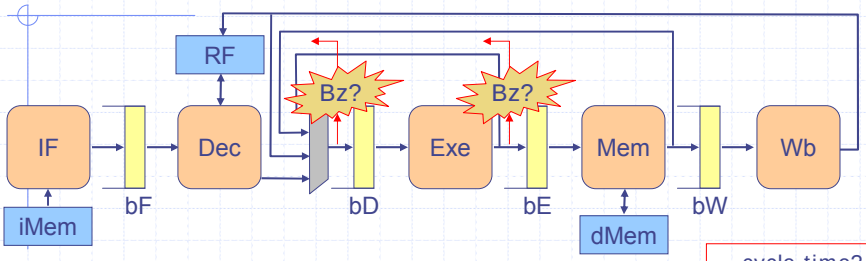
Performance Specifications

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Simple processor pipeline

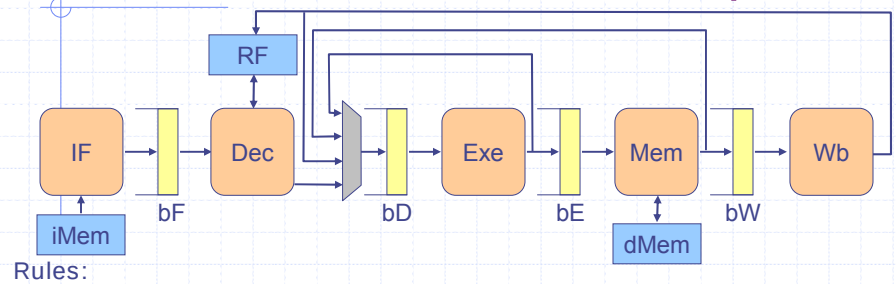


- Functional behavior is well understood
- Intuition about performance is lacking
  - Should the branch be resolved in the Decode or Execute stage?
  - Should the branch target address be latched before its use?
- Experimentation is required to evaluate design alternatives

cycle time?
area?
execution time?

We present a design flow that makes such experimentation easy for the designer

Need for Performance Specs



Rules:

F = {Fetch}	D = { DecAdd, DecBz, ... }	E = { ExeAdd, ExeBzTaken, ExeBzNotTaken, ... }	M = { MemLd, MemSt, MemWB, ...}	W = {Wb}

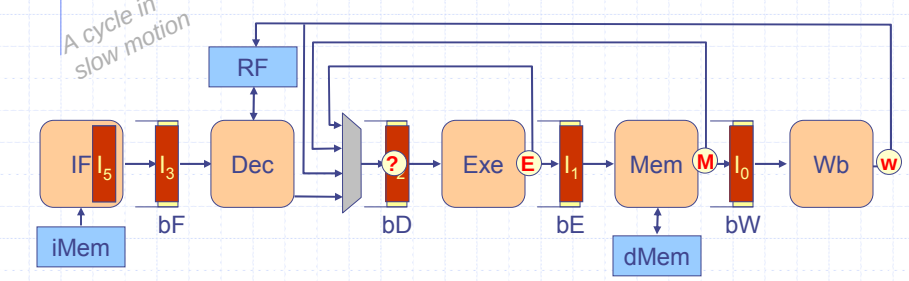
- What is the design's performance / throughput?
- Reference model implies one rule per cycle execution

Designer's goal is usually different and based on the application!

Pipelining via Performance specification

- The designer wants a pipeline which executes one instruction every cycle
- Performance spec for a pipelined processor:

$$W < M < E < D < F$$



More Performance Specification

F = {Fetch}	D = { DecAdd, DecBz, ... }	E = { ExeAdd, ExeBzTaken, ExeBzNotTaken, ... }	M = { MemLd, MemSt, MemWB, ...}	W = {Wb}

We allow the designer to specify performance!

W < M < E < D < F ≡ *pipelined*

1) W < M < E* < D < F
2) W < M < *ExeBzTaken* ≡ *pipelined except for ExeBzTaken*

What do the following mean?

F < D < E < M < W ≡ *unpipelined* *(assuming buffers start empty)*

W < W < M < M < E < E < D < D < F < F ≡ *two-way superscalar!*

Synthesis algorithms ensure that performance specs are satisfied and guarantee that functionality is not altered.

Why is functionality maintained?

- A few observations about rule-based systems:
  - Adding a new rule to a system can only introduce new behaviors
  - If the new rule is a *derived* rule, then it does not add new behaviors
- Composed rules:
  - Given rules:

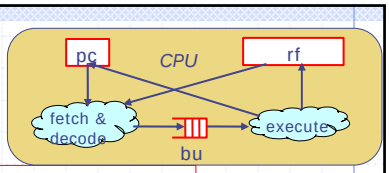
R_a: when $\pi_a(s)$ => s := $\delta_a(s)$;

R_b: when $\pi_b(s)$ => s := $\delta_b(s)$;

  - The composed rule is a derived rule:

$R_{a,b}$: when $\pi_a(s)$ & $\pi_b(\delta_a(s))$ => s := $\delta_b(\delta_a(s))$;

Scheduling Specifications




```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule
```

```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 rf.upd(rd, va+vb); bu.deq(); endrule
rule execBzTaken(it matches tagged Bz {cond:.cv,addr:.av}
                &&& (cv == 0));
      pc <= av; bu.clear(); endrule
rule execBzNotTaken(it matches tagged Bz {cond:.cv,addr:.av}
                &&& !(cv == 0));
      bu.deq(); endrule
rule execLoad(it matches tagged ELoad{dst:.rd,addr:.av});
  rf.upd(rd, dMem.read(av)); bu.deq(); endrule
rule execStore(it matches tagged EStore{val
   dMem.write(av, vv); bu.deq(); endrule
```

execAdd < fetch

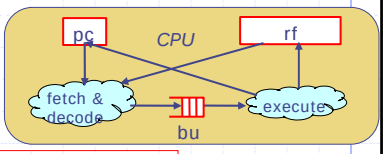
execBzTaken < fetch

execBzNotTaken < fetch ?

execLoad < fetch

execStore < fetch

Implications for modules

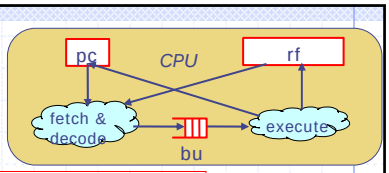



```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule
```

```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 rf.upd(rd, va+vb); bu.deq();
endrule
```

- execAdd < fetch ⇒
  - rf: sub > upd
  - bu: {find, enq} > {first , deq}

Branch rules




```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule
```

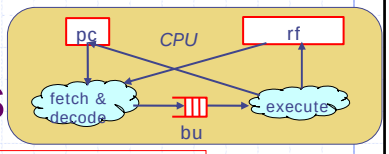
```
rule execBzTaken(it matches tagged Bz {cond:.cv,addr:.av}
              &&& (cv == 0));
     pc <= av; bu.clear(); endrule

rule execBzNotTaken(it matches tagged Bz {cond:.cv,addr:.av}
              &&& !(cv == 0));
     bu.deq(); endrule
```

- execBzTaken < fetch ?
  - Should be treated as conflict – give priority to execBzTaken
- execBzNotTaken < fetch

  bu: {first , deq} < {find, enq}

Load-Store Rules




```
rule fetch_and_decode (!stallfunc(instr, bu));
        bu.enq(newIt(instr,rf));
        pc <= predIa;
endrule
```

```
rule execLoad(it matches tagged ELoad{dst:.rd,addr:.av});
  rf.upd(rd, dMem.read(av)); bu.deq();
endrule
```

```
rule execStore(it matches tagged EStore{value:.vv,addr:.av});
   dMem.write(av, vv); bu.deq();
endrule
```

- execLoad < fetch ?
  - Same as execAdd, i.e.,

    rf: upd < sub

    bu: {first , deq} < {find, enq}
- execStore < fetch ?

  bu: {first , deq} < {find, enq}

Properties Required of Register File & FIFO to meet performance specs

- Register File:
  - rf.upd < rf.sub
- FIFO
  - bu: {first , deq} < {find, enq} ⇒
    - bu.first < bu.find
    - bu.first < bu.enq
    - bu.deq < bu.find
    - bu.deq < bu.enq

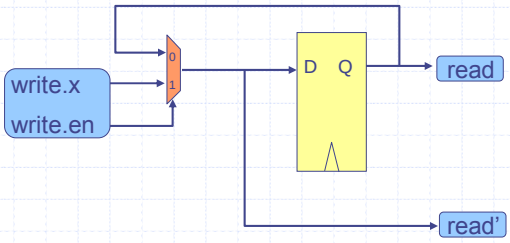
The good news …

- It is always possible to transform your design to meet desired concurrency and functionality
  - Though critical path and hence the clock period may increase

Register Interfaces

read < write

write < read ?

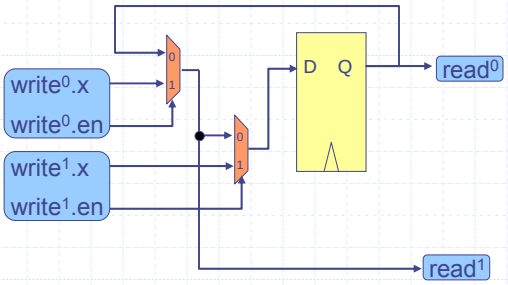


read' – returns the current state when write *is not enabled*
read' – returns the value being written if write *is enabled*

Ephemeral History Register (EHR)

[Rosenband MEMOCODE'04]

$read^0 < write^0 < read^1 < write^1 < ....$



$write^{i+1}$ takes precedence over $write^i$

Transformation for Performance

execAdd < fetch
execBzTaken < fetch
execLoad < fetch
execStore < fetch

```
rule fetch_and_decode (!stallfunc[1](instr, bu));
        bu.enq[1](newIt(instr,rf));
        pc <= predIa;
endrule
```

```
rule execAdd
  (it matches tagged EAdd{dst:.rd,src1:.va,src2:.vb});
 rf.upd[0](rd, va+vb); bu.deq[0](); endrule
rule execBzTaken(it matches tagged Bz {cond:.cv,addr:.av}
                 &&& (cv == 0));
      pc <= av; bu.clear(); endrule
rule execBzNotTaken(it matches tagged Bz {cond:.cv,addr:.av}
                 &&& !(cv == 0));
      bu.deq[0](); endrule
rule execLoad(it matches tagged ELoad{dst:.rd,addr:.av});
  rf.upd[0](rd, dMem.read(av)); bu.deq[0](); endrule
rule execStore(it matches tagged EStore{value:.vv,addr:.av});
   dMem.write(av, vv); bu.deq[0](); endrule
```


One Element FIFO *using EHRs*

```
module mkFIFO1 (FIFO#(t));
  EHReg2#(t)    data  <- mkEHReg2U();
  EHReg2#(Bool) full  <- mkEHReg2(False);
  method Action enq[0](t x) if (!full.read[0]);
    full.write[0] <= True;  data.write[0] <= x;
  endmethod
  method Action deq[0]() if (full.read[0]);
    full.write[0] <= False;
  endmethod
  method t first[0]() if (full.read[0]);
    return (data.read[0]);
  endmethod
  method Action clear[0]();
    full.write[0] <= False;
  endmethod
endmodule
```

first[0] < deq[0] < enq[1]

```
method Action enq[1](t x) if (!full.read[1]);
  full.write[1] <= True; data.write[1] <= x;
endmethod
```

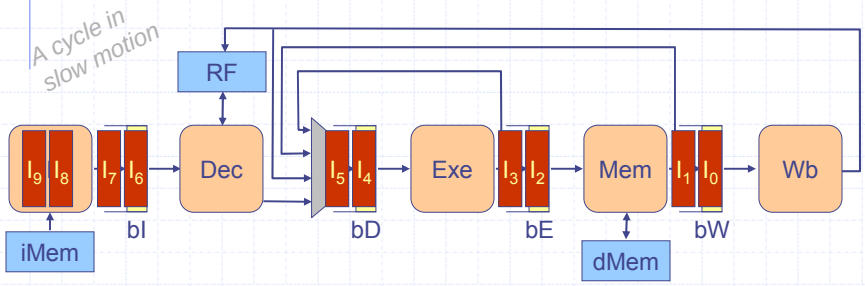
Experiments in scheduling

Dan Rosenband, ICCAD 2005

- What happens if the user specifies:

Wb < Wb < Mem < Mem < Exe < Exe < Dec < Dec < IF < IF

No change in rules *a superscalar processor!*



Executing 2 instructions per cycle requires more resources but is functionally equivalent to the original design

4-Stage Processor Results

Design	Benchmark (cycles)	Area 10ns (μm^2)	Timing 10ns (ns)	Area 2ns (μm^2)	Timing 2ns (ns)
1 element fifo:					
No Spec	18525	24762	5.85	26632	2.00
Spec 1	11115	25094	6.83	33360	2.00
Spec 2	11115	25264	6.78	34099	2.04
2 element fifo:					
No Spec.	18525	32240	7.38	39033	2.00
Spec 1	11115	32535	8.38	47084	2.63
Spec 2	7410	45296	9.99	62649	4.72

benchmark: a program containing additions / jumps / loadc's

Dan Rosenband & Arvind 2004

Summary

- For most designs BSV Compiler does good scheduling of rules with some user annotations for priority
- However, for complex designs sometimes concurrency control is quite difficult and requires a good understanding on the part of the designer of the concurrency issues
- Performance specification is a good, safe solution but is not implemented in the compiler yet.
  - user can do manual "renaming" and use EHRs to meet most performance goals
- RWires can solve any problems but exacerbate the correctness issue
- Synchronous pipelines (single rule) can avoid many problems but is not recommended for complex designs